# SAVITCH ABSOLUTE JAVA

# Name: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Seat # \_\_\_\_

**Homework 5 - On Lecture 6 - Polymorphism**

**(100 points) Hours:**

**The homework is to be turned in as a *PAPER AND PENCIL i.e., HANDWRITTEN ANSWER ONLY!(with your terminal #!)* in the first ten minutes of the due date class.**

**Also an implementation in NetBeans Java is ALSO required, thus you are to submit the ZIPPED project to BB and download it in 232 PGH the first 10 minutes of class. Hardcopy with screenshots of the running program and the SOURCE CODE are also needed.**

**TURNING IN THE HOMEWORK INSTRUCTIONS will be PENALTY OF -10 points.**

**I UNDERSTAND THAT TURNING ANOTHER’s WORK IN is CHEATING.**

**I UNDERSTAND THAT ANY KIND OF DISSEMINATION of this WORK is CHEATING.**

**I CERTIFY THAT THE HOMEWORKs SOLUTIONs ARE MY OWN WORK!**

**?**

**X**

**V**

**SIGNATURE:**

**HOMEWORK CHECKLIST (YOU MUST GRADE YOURSELF!):**

**TA check, is Homework5.doc & Homework5.zip**

**in BB?**

1. **DID TURN IN HOMEWORK INSTRUCTIONS?\* -10 points**
2. **1.? 70 points**
3. **2.? H & E (attach to BB) – WORD ONLY 30 points**
4. **Homework5.2 JAVA.zip NOT submitted to BB?\* -20 points**
5. **Homework5.2 JAVA.zip NOT running in class?\* -20 points**

**\* If NOT, do not enter anything in the box!**

TA **grade or check**

**Screenshot?**

**PLEASE ENTER YOUR GRADE IN THIS BOX:**

**By Hand?**

# I understand that if the .zip file is NOT in BB and I did not check the BOX, I will get a ZERO for the Homework! 1. (70 pts)

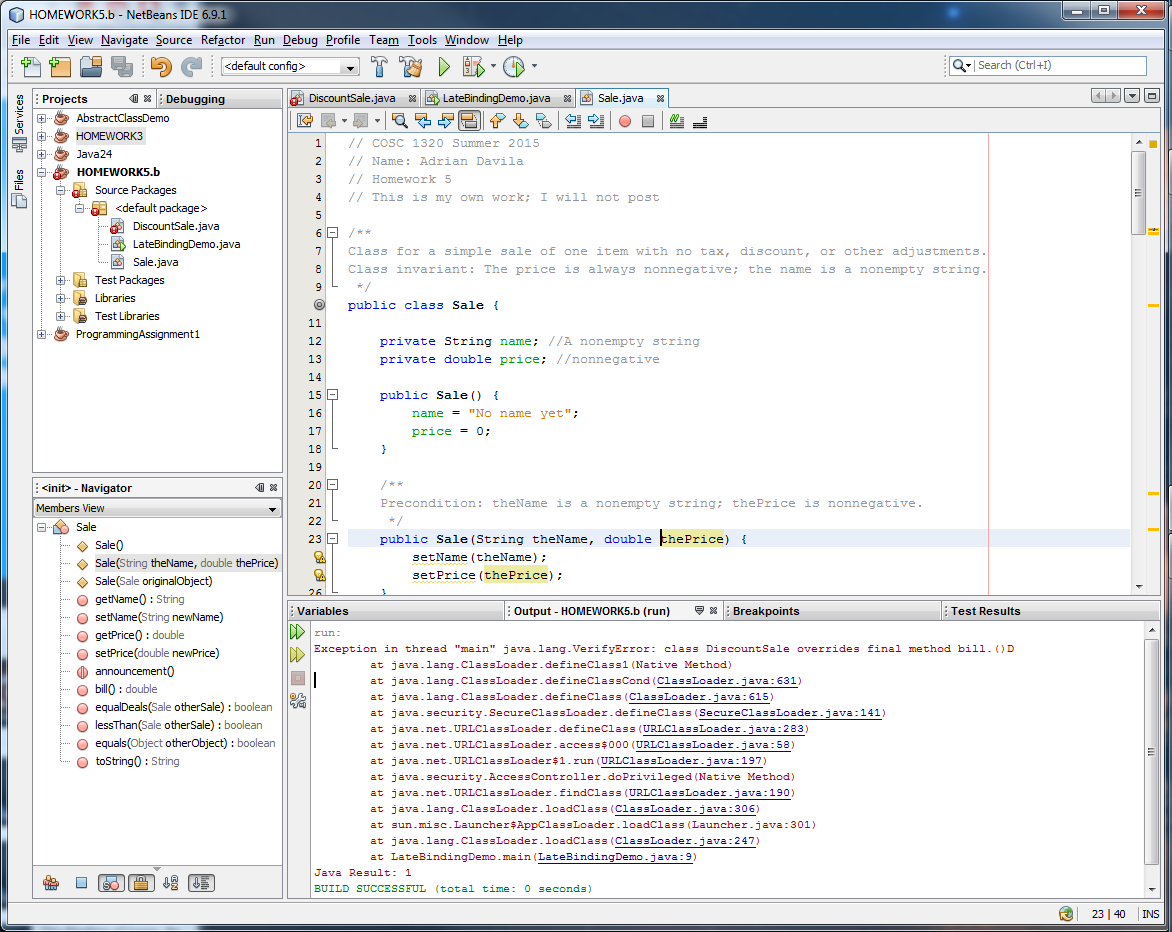
a. (5 pts)Explain the difference between the terms late binding and polymorphism.

**ANSWER:**

b. (5 pts)Suppose you modify the definitions of the class Sale (**Class Participation A; LateBindingDemo**) by adding the modifier final to the definition of the method bill. How would that change the output of the program?

**RERUN the LateBindingDemo program in Netbeans AS HOMEWORK5.b build and run and show the output!**

**ANSWER:**



c. (5 pts)Would it be legal to add the following method definition to the class DiscountSale?

public **static** boolean isAGoodBuy( Sale theSale)

{

return ( theSale. getDiscount() > 20);

}

**ANSWER:**

d. (5 pts)Consider the following code, which is identical to the code discussed in the opening of the subsection, “ Downcasting and Upcasting,” except that we have added the type cast shown in color:

Sale saleVariable;

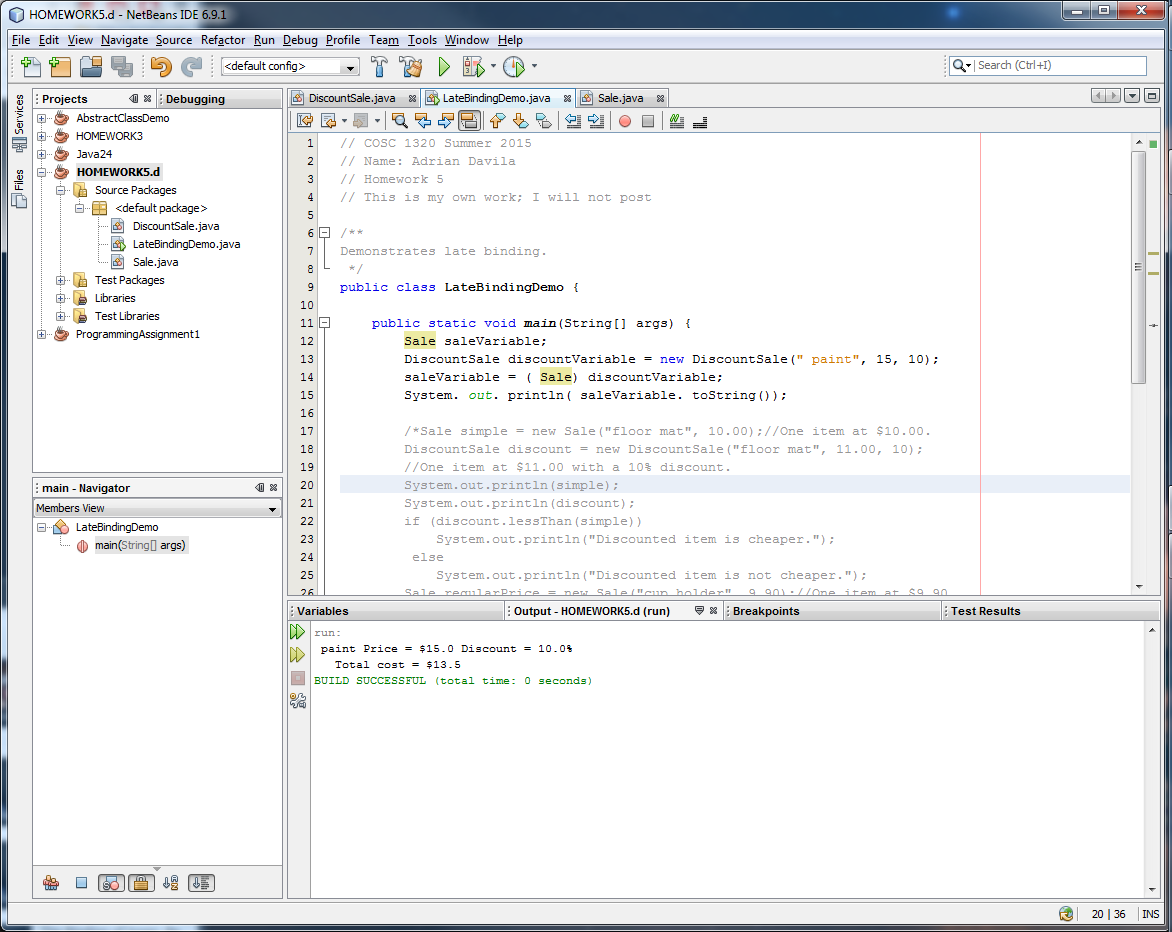
DiscountSale discountVariable = **new** DiscountSale(" paint", 15, 10);

saleVariable = **( Sale)** discountVariable;

System. out. println( saleVariable. toString());

We saw that without the type cast, the definition of the toString method used is the one given in the definition of the class DiscountSale. With this added type cast, will the definition of the toString method used still be the one given in DiscountSale or will it be the one given in the definition of Sale? **RERUN the LateBindingDemo program in Netbeans AS HOMEWORK5.d build and run and show the output!**

**ANSWER:**



e. (5 pts)Would it be legal to add the following method definition to the class DiscountSale? What about adding it to the class Sale?

public static void showDiscount( Sale object)

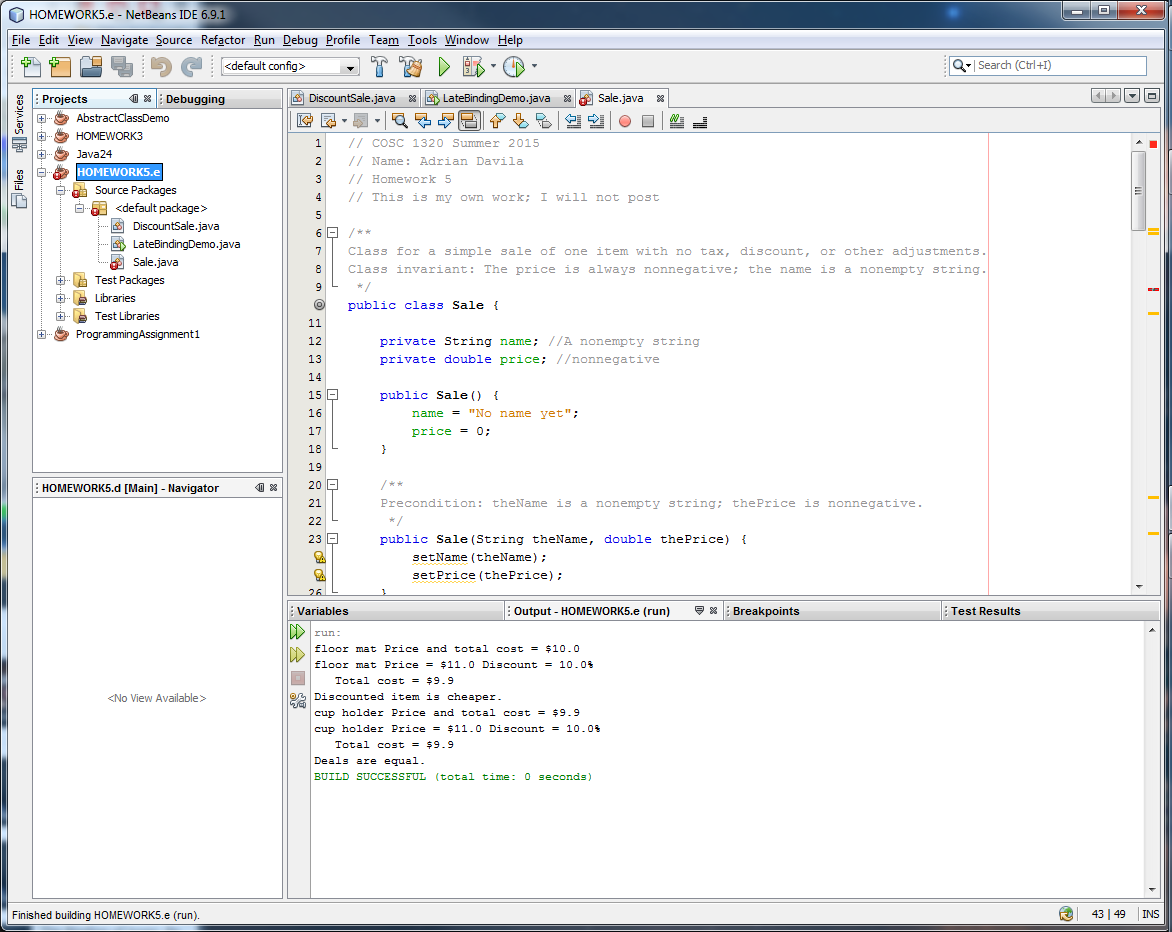
{

System. out. println(" Discount = " + object. getDiscount());

}

**RERUN the LateBindingDemo program in Netbeans AS HOMEWORK5.e build and run and show the output!**

**ANSWER:**



f. (5 pts)What output is produced by the following code?

Sale someObject = new DiscountSale(" map", 5, 0);

DiscountSale ds = new DiscountSale();

if ( someObject instanceof DiscountSale)

{

ds = ( DiscountSale) someObject;

System. out. println(" ds was changed to " + someObject);

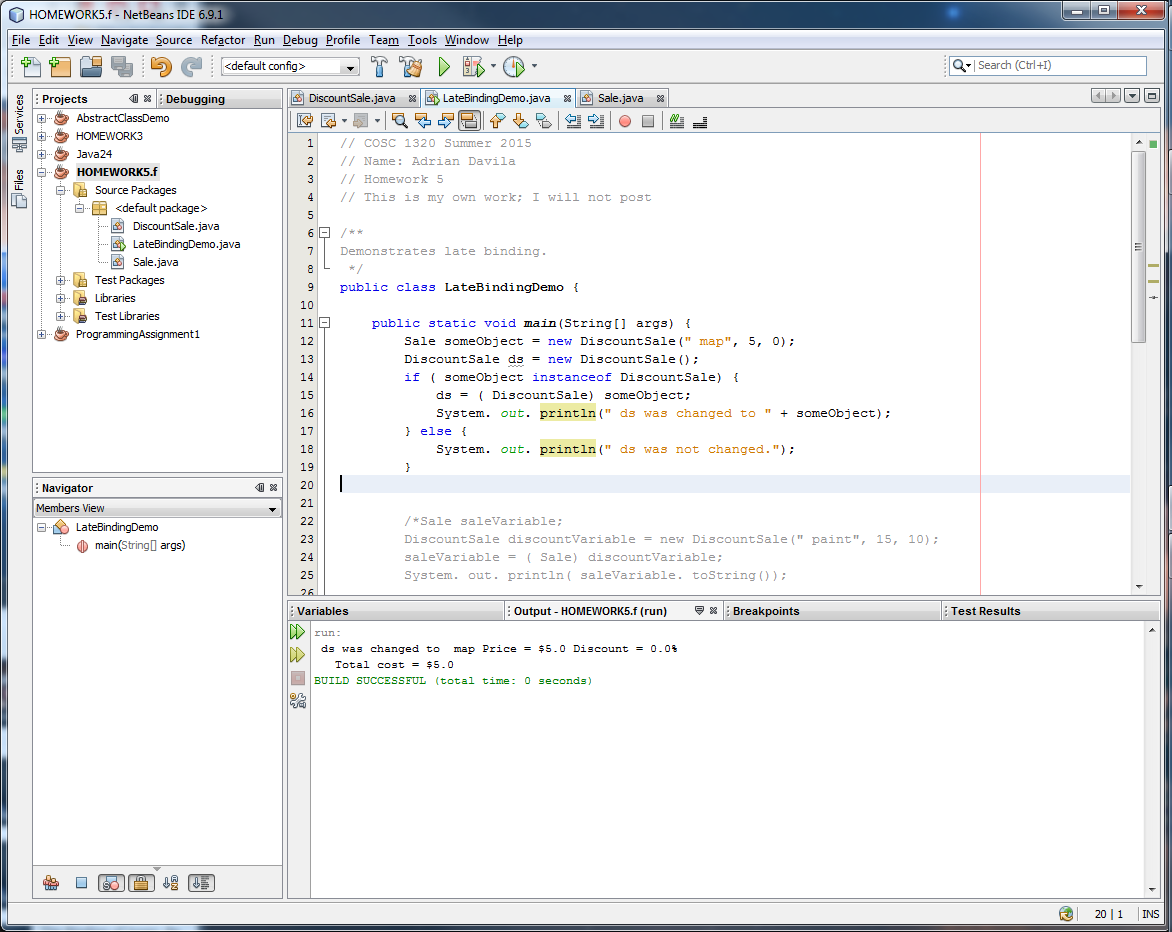
}

else

System. out. println(" ds was not changed.");

**RERUN the LateBindingDemo program in Netbeans AS HOMEWORK5.f build and run and show the output!**

**ANSWER:**



g. (5 pts) **What** output is produced by the following code?

Sale someObject = new Sale(" map", 5);

DiscountSale ds = new DiscountSale();

if ( someObject instanceof DiscountSale)

{

ds = ( DiscountSale) someObject; System. out.

println(" ds was changed to " + someObject);

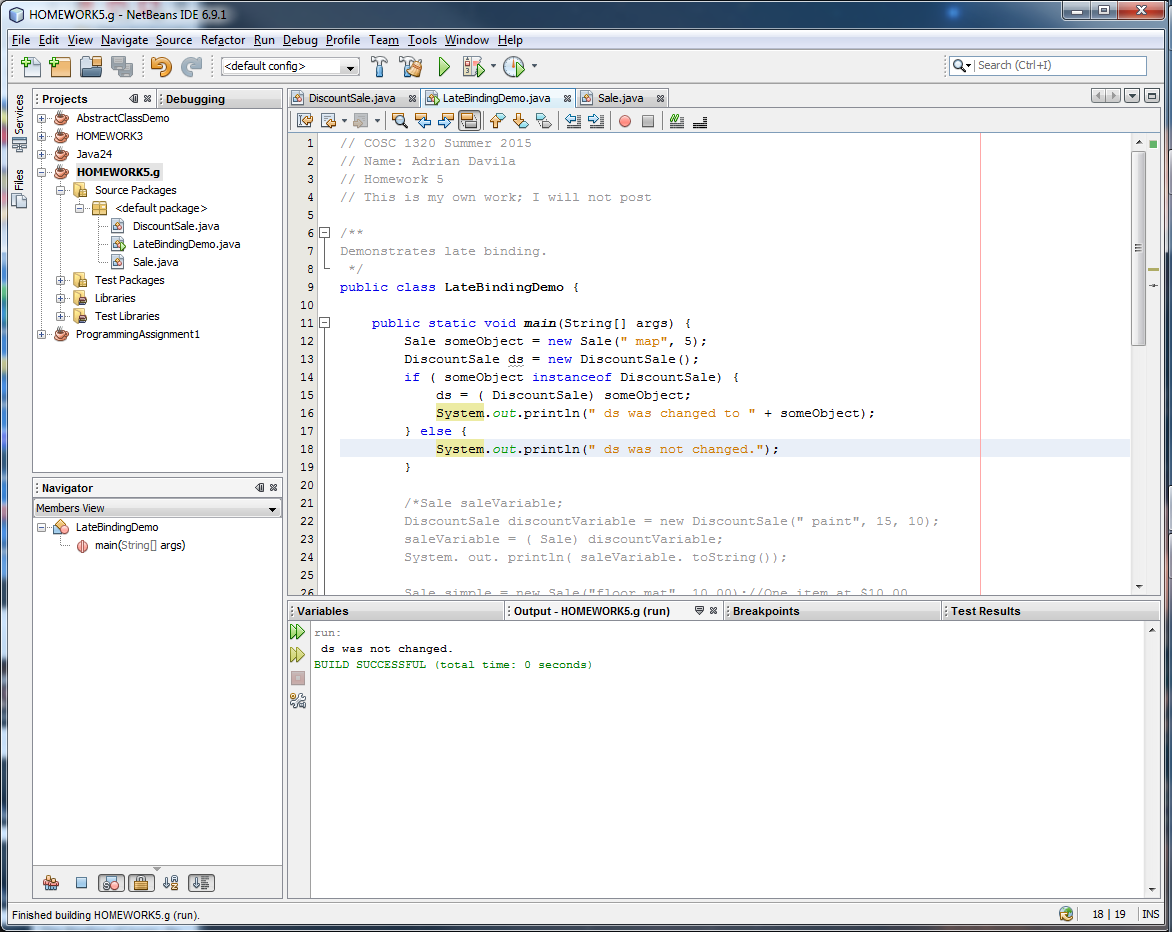
}

else

System. out. println(" ds was not changed.");

**RERUN the LateBindingDemo program in Netbeans AS HOMEWORK5.g build and run and show the output!**

**ANSWER:**



h. (5 pts)Suppose we removed the qualifier static from the method announcement() in both Sale and DiscountSale .

What would be the output produced by the following code?

Sale s = new Sale( );

DiscountSale discount = new DiscountSale();

s. announcement( );

discount. announcement( );

System. out. println(" No surprises so far, but wait.");

Sale discount2 = discount;

System. out. println( " discount2 is a DiscountSale object in a Sale variable.");

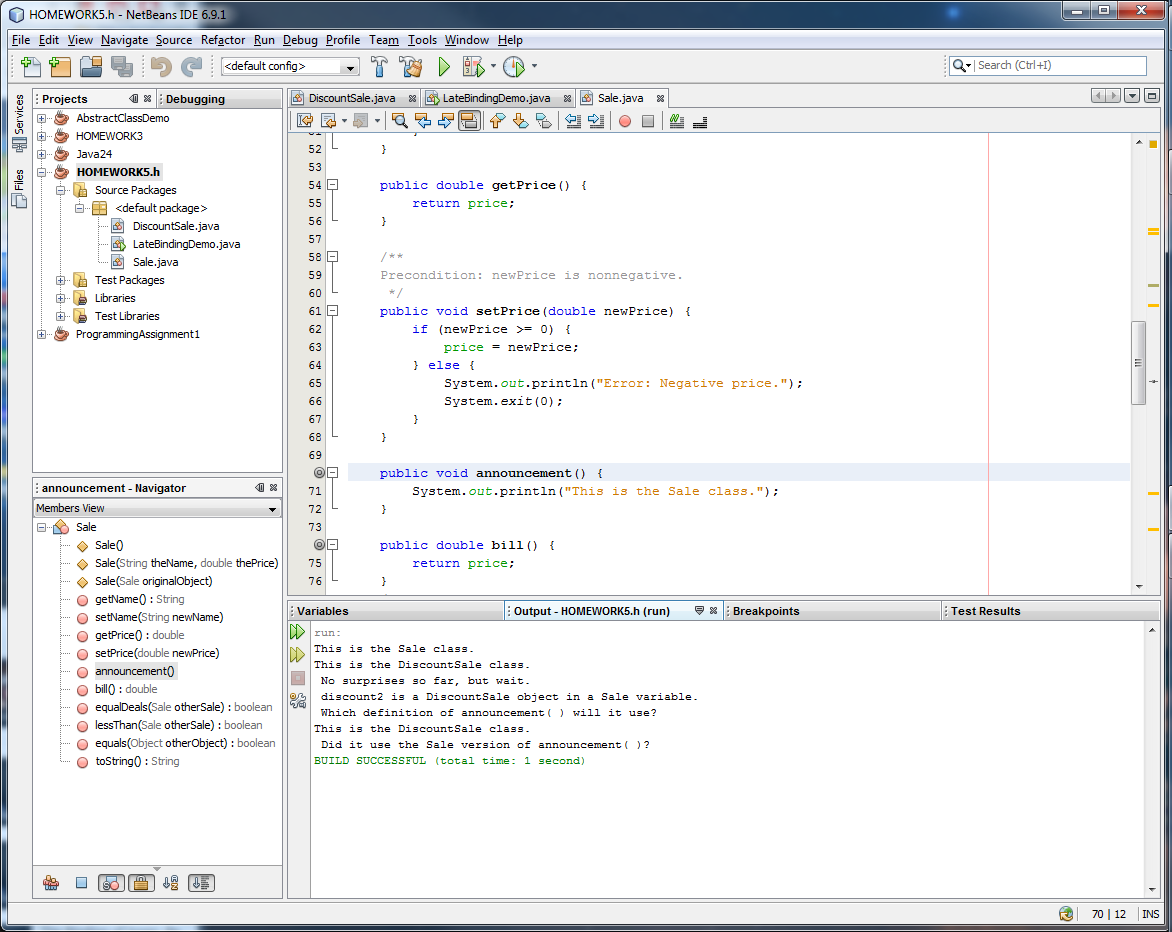
System. out. println( " Which definition of announcement( ) will it use?");

discount2. announcement( );

System. out. println( " Did it use the Sale version of announcement( )?");

**RERUN the LateBindingDemo program in Netbeans AS HOMEWORK5.h build and run and show the output!**

**ANSWER:**



(**BY HAND**)

i. (5 pts)Can a method definition include an invocation of an abstract method?

**ANSWER:**

j. (5 pts)Can you have a variable whose type is an abstract class?

**ANSWER:**

k. (5 pts)Can you have a parameter whose type is an abstract class?

**ANSWER:**

l. (5 pts)Is it legal to have an abstract class in which all methods are abstract?

**ANSWER:**

m. (5 pts)The abstract class Employee uses the method definitions as described in **Class Participation C; AbstractClassDemo**. Given the following improved version of equals:

public boolean equals( Object otherObject)

{

if ( otherObject == null)

return false;

else if ( getClass() != otherObject. getClass())

return false;

else

{

Employee otherEmployee = ( Employee) otherObject;

return ( name. equals( otherEmployee. name) && hireDate. equals(

otherEmployee. hireDate));

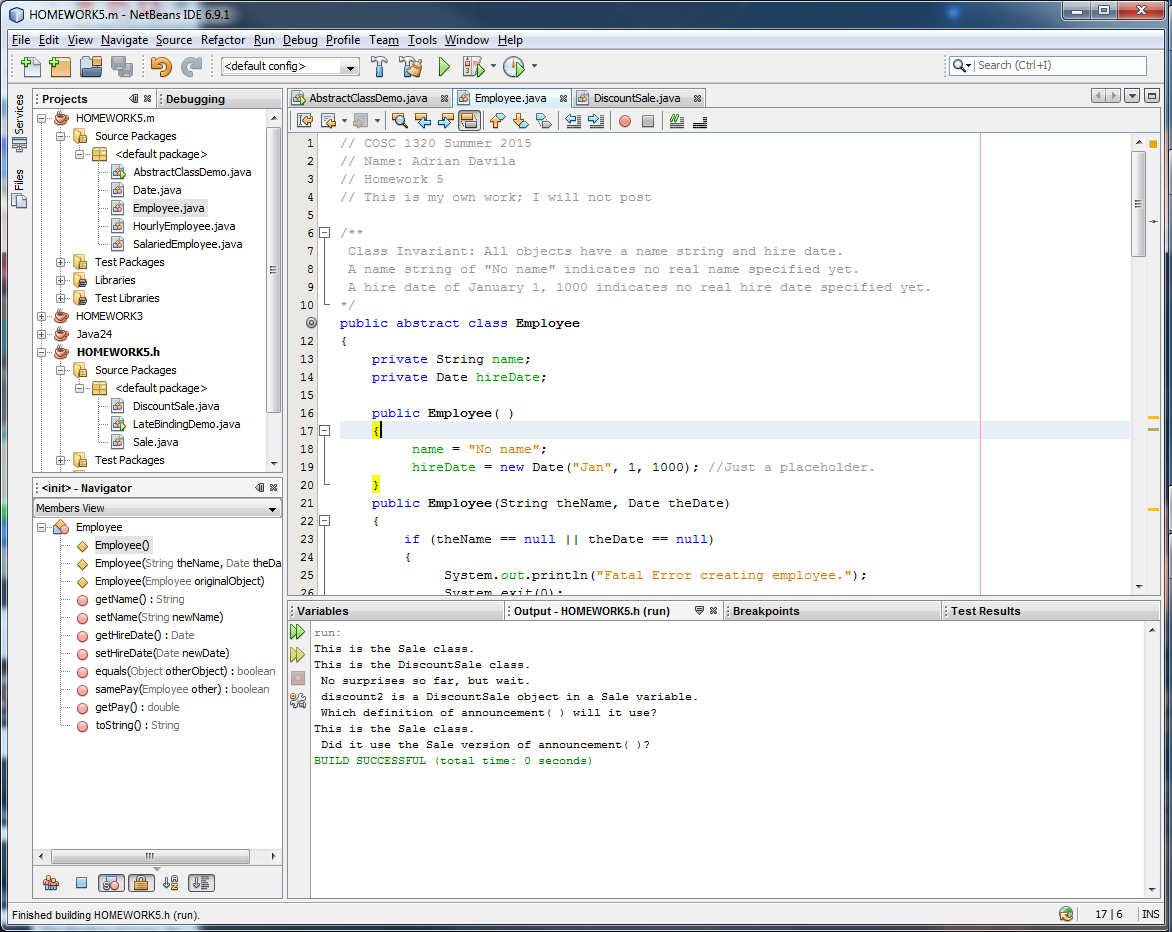
}

}

Would it be legal to replace the version of equals for the abstract class Employee with this improved version?

**RERUN the AbstractClassDemo program in Netbeans AS HOMEWORK5.m build and run and show the output!**

**ANSWER:**



n. (5 pts)The abstract class Employee described in **Class Participation C; AbstractClassDemo** has a constructor ( in fact, it has more than one). But using a constructor to create an instance of an abstract class, as in the following, is illegal:

Employee joe = new Employee(); // Illegal

So why bother to have any constructors in an abstract class? Aren’t they useless?

**ANSWER:**

**2.** (30 pts) **UML Class Diagram** (**MICROSOFT WORD; Textual Analysis – TA Cut&Paste&Rearrange**).

In Programming Project 3 from Chapter 7 the Alien Class was rewritten to use **inheritance**. The rewritten Alien Class should be made **abstract** since there will never be a need to create an instance of it, only its derived Classes. Change this to an **abstract** Class and also make the getDamage method an **abstract** method. Test the Class from your main method to ensure that it still operates as expected.

a. (10 pts) **OOA & OOD: Draw the UML Class Diagram.**

**ANSWER:**

Any DIAGRAM that is NOT the result of CUT and PASTE

WILL BE IGNORED (YOU WILL GET ZERO POINTS)

b. (20 pts) **Implementation: NetBeans AS HOMEWORK5.2 build and run, Source Code and Screenshot. Submit zipped project to BB!**

<http://media.pearsoncmg.com/aw/aw_savitch_abjava_4/videos/video8_2/video.html>

Project 3 Chapter 7:

*/\*\**

*\* Question3Alien.java*

*\**

*\* This program introduces inheritance by rewriting a non-OO class*

*\* using a switch statement for types into one with derived classes*

*\* for the subtypes. A getDamage method returns the amount of*

*\* damage that an alien inflicts. This design is superior to the*

*\* old one since it is now independent to add a new type of Alien*

*\* to the game; we no longer need to change code in the Alien class.*

*\**

*\* Created: Sat Mar 19 2005*

*\**

*\* @author Kenrick Mock*

*\* @version 1*

*\*/*

*Alien*

*Alien:* DEFAULT\_DAMAGE: int //1

*public abstract class Alien {*

*public static final int DEFAULT\_DAMAGE = 0;*

*private int health; // 0=dead, 100=full strength*

*Alien:*health: int //2

*private String name;*

*Alien:* name: String //3

*/\*\**

*\* Default constructor; Initialize name and health.*

*\*/*

*public Alien()*

*Alien:* Alien() //1

*{*

*health = 100;*

*name = "";*

*}*

*/\*\**

*\* Default constructor; Initialize variables to input parameters.*

*\* @param health Initial health*

*\* @param name Initial name*

*\*/*

*public Alien(int health, String name)*

*{*

*Alien*: Alien(int, String) //2

*this.health = health;*

*this.name = name;*

*}*

*// ======================*

*// Various accessor and mutator methods*

*// ======================*

*public int getHealth()*

*Alien*: getHealth () //3

*{*

*return health;*

*}*

*public void setHealth(int health)*

*{*

*Alien*: setHealth (int) //4

*this.health = health;*

*}*

*public String getName()*

*Alien*: getName (5) //5

*{*

*return name;*

*}*

*public void setName(String name)*

*{*

*Alien*: setName (String) //6

*this.name = name;*

*}*

*/\*\**

*\* getDamage returns the amount of damage this alien inflicts.*

*\*/*

*public abstract int getDamage()*

*Alien:* *getDamage*(): int //7

*{*

*return DEFAULT\_DAMAGE;*

*}*

*} // Alien*

*/\*\* The SnakeAlien class extends the Alien class*

*\* since it is a sub-type of Alien. By doing this for*

*\* all the alien types, we can eliminate the "type" variable*

*\* in the original Alien class.*

*\*/*

SnakeAlien: SNAKE\_DAMAGE: int //1

*class SnakeAlien extends Alien*

*{*

*public static final int SNAKE\_DAMAGE = 10;*

*/\*\**

*\* Constructors*

*\*/*

*public SnakeAlien()*

SnakeAlien: SnakeAlien(): //1

*{*

*super();*

*}*

SnakeAlien:SnakeAlien(int, String):void //2

*public SnakeAlien(int health, String name)*

*{*

*super(health,name);*

*}*

*/\*\**

*\* getDamage returns the amount of damage this alien inflicts.*

*\*/*

*public int getDamage()*

SnakeAlien:getDamage (): int //3

*{*

*return SNAKE\_DAMAGE;*

*}*

*} // SnakeAlien*

*/\*\**

*\* Class for the Ogre Alien*

*\*/*

OgreAlien: OGRE\_DAMAGE: int //1

*class OgreAlien extends Alien*

*{*

*public static final int OGRE\_DAMAGE = 6;*

*/\*\**

*\* Constructors*

*\*/*

*public OgreAlien()*

OgreAlien: OgreAlien():void //1

*{*

*super();*

OgreAlien: OgreAlien(int, String):void //2

*}*

*public OgreAlien(int health, String name)*

*{*

*super(health,name);*

*}*

*/\*\**

*\* getDamage returns the amount of damage this alien inflicts.*

*\*/*

*public int getDamage()*

OgreAlien: *getDamage* ():int //3

*{*

*return OGRE\_DAMAGE;*

*}*

*} // OgreAlien*

*/\*\**

*\* Class for the Marshmallow Alien*

*\*/*

*class MarshmallowAlien extends Alien*

*MarshmallowAlien*: MARSHMALLOW\_DAMAGE: int //1

*{*

*public static final int MARSHMALLOW\_DAMAGE = 1;*

*/\*\**

*\* Constructors*

MarshmallowAlien: MarshmallowAlien(): void//1

*\*/*

*public MarshmallowAlien()*

*{*

*super();*

*}*

MarshmallowAlien: MarshmallowAlien(int, String): void//2

*public MarshmallowAlien(int health, String name)*

*{*

*super(health,name);*

*}*

*/\*\**

*\* getDamage returns the amount of damage this alien inflicts.*

*\*/*

MarshmallowAlien: getDamage (): void //3

*public int getDamage()*

*{*

*return MARSHMALLOW\_DAMAGE;*

*}*

*} // MarshmallowAlien*

*/\*\**

*\* This class stores an array of Aliens that comprise a "pack"*

*\*/*

AlienPack: aliens: Alien[] //1

*class AlienPack*

*{*

*private Alien[] aliens;*

AlienPack: AlienPack():void //1

*public AlienPack(int numAliens)*

*{*

*aliens = new Alien[numAliens];*

*}*

AlienPack: AlienPack(Alien, int):void //2

*public void addAlien(Alien newAlien, int index)*

*{*

*aliens[index] = newAlien;*

*}*

AlienPack: *getAliens* (): Alien[] //3

*public Alien[] getAliens()*

*{*

*return aliens;*

*}*

*/\*\**

*\* To calculate the damage inflicted by all aliens in the pack*

*\* we can now simply iterate through each alien and call its*

*\* getDamage() method*

AlienPack: calculateDamage (): int //4

*\*/*

*public int calculateDamage()*

*{*

*int damage = 0;*

*for (int i=0; i < aliens.length; i++)*

*{*

*damage += aliens[i].getDamage();*

*}*

*return damage;*

*}*

*} // AlienPack*

*class Question3Alien*

*{*

*// ======================*

*// main method.*

*// In main we simply create a small pack and output its total damage (17).*

*// ======================*

*public static void main(String[] args)*

*{*

*OgreAlien brutus = new OgreAlien(100,"brutus");*

*SnakeAlien slimy = new SnakeAlien(100,"slimy");*

*MarshmallowAlien puffy = new MarshmallowAlien(100,"puffy");*

*AlienPack pack = new AlienPack(3); // 3 aliens in the pack*

*pack.addAlien(brutus, 0);*

*pack.addAlien(slimy, 1);*

*pack.addAlien(puffy, 2);*

*System.out.println("Total pack damage = " + pack.calculateDamage());*

*}*

*} // Question3Alien*

**ANSWER:**

|  |
| --- |
| *Alien* |
| *Alien:* DEFAULT\_DAMAGE: int //1  *Alien:*health: int //2  *Alien:* name: String //3 |
| *Alien:* Alien() //1  *Alien*: Alien(int, String) //2  *Alien*: getHealth () //3  *Alien*: setHealth (int) //4  *Alien*: getName (5) //5  *Alien*: setName (String) //6  *Alien:* *getDamage*(): int //7 |

|  |
| --- |
| Question3Alien |
|  |
| Question3Alien: main(String[] args): void //1 |

|  |
| --- |
| OgreAlien |
| OgreAlien: OGRE\_DAMAGE: int //1 |
| OgreAlien: OgreAlien():void //1  OgreAlien: OgreAlien(int, String):void //2  OgreAlien: *getDamage* ():int //3 |

|  |
| --- |
| MarshmallowAlien |
| *MarshmallowAlien*: MARSHMALLOW\_DAMAGE: int //1 |
| MarshmallowAlien: MarshmallowAlien(): void//1  MarshmallowAlien: MarshmallowAlien(int, String): void//2  MarshmallowAlien: getDamage (): void //3 |

|  |
| --- |
| AlienPack |
| AlienPack: aliens: Alien[] //1 |
| AlienPack: AlienPack():void //1  AlienPack: AlienPack(Alien, int):void //2  AlienPack: *getAliens* (): Alien[] //3  AlienPack: calculateDamage (): int //4 |

|  |
| --- |
| SnakeAlien |
| SnakeAlien: SNAKE\_DAMAGE: int //1 |
| SnakeAlien: SnakeAlien(): //1  SnakeAlien:SnakeAlien(int, String):void //2  SnakeAlien:getDamage (): int //3 |